FEATURE
ARTICLE

Walter Banks

Fuzzy Concepts Using C

Walter takes fuzzy logic into the trenches with this down-to-earth implementation of fuzzy logic using C on small micros. He uses examples from real-world applications to illustrate his points.

Much of the power of fuzzy logic comes from its ability to focus on specific areas of interest. In many cases, fuzzy-logic–based systems can produce superior results with significantly lower resolution than congenital approaches to problem solving.

A lot has been written about fuzzy logic as another programming paradigm, but fuzzy logic is now regularly mixed with conventional code in many applications. Fuzzy-logic operators are the formal method of manipulating linguistic variables.

This article is primarily aimed at the eight-bit embedded-system developer who is using a high-level language to implement an application. The notation in the fuzzy examples uses the syntax of Fuzz-C, a C preprocessor that translates mixed fuzzy and C into C.

There is no need or reason (other than convenience) to use this preprocessor. The examples show the direct relationship between fuzzy expressions and C. My main point: fuzzy logic doesn't need special hardware or software, and it offers a powerful tool for problem solving.

Central to fuzzy-logic manipulation are linguistic variables. Linguistic variables are nonprecise variables that convey information.

I can say, for example, that I drove to work fast or that I drove at the spee limit. The first description depicts behavior that borders on reckless. The second portrays me driving a profile o time, space, and different speeds on the expressway and residential roads.

Depending on context, driving at th speed limit can have other interpretations as well. If the expressway speed limit is 55 MPH, then it might mean that I'm driving close to 55 MPH—perhaps 52 or 58 with some distribution around 55. My attention to the exact number may be changed by oth factors. Known police-radar locations often limit my speed to an exact 55!

Linguistic variables in a computer require there to be a formal way of describing the linguistic variable in crisp terms the computer can deal with. The graph in Figure 1 shows th relationship between measured speed and the linguistic term FAST.

Each of us may differ about what counts as fast. But, at some speed, we all say that it is not fast, and at some other point, we agree that it is fast.

In the space between fast and not fast, the speed is, to some degree, bot The horizontal axis in Figure 1 shows the measured or crisp value of speed. The vertical axis describes the degree to which a linguistic variable fits the crisp measured data.

I can describe temperature in a no graphical way with the declaration:

```
LINGUISTIC Speed TYPE unsign
 int MIN 0 MAX 100 {MEMBER FA
  {60, 80, 100, 100} };
```

This declaration describes both the crisp variable Speed as an unsigned i

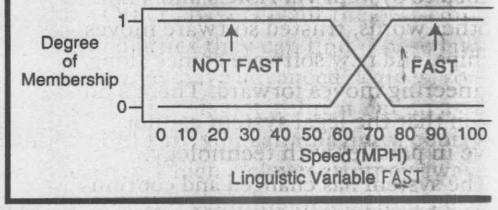


Figure 1—*Here the linguistic variable FAST is compared to t crisp speed, producing a degree of membership.*

```
unsigned int Speed;   /* Crisp value of Speed */
unsigned char Speed_FAST (unsigned int __CRISP) {
  if (__CRISP < 60) return(0);
  else{
    if (__CRISP <= 80) return(((__CRISP - 60) * 12) + 7);
    else {
      return(255); } } }
```

and a linguistic member FAST as a trapezoid with specific parameters.

If I add the linguistic variable FAST to a computer program running in an embedded controller, I need to be able to translate the graphical representation into code. The C code fragment in Listing 1 is an example of how this might be done.

The function Speed_FAST returns a degree of membership scaled between 0 and 255. This type of simple calculation is the first tool required for calculations of fuzzy-logic operations.

The same code can be translated to run on many different embedded micros. The generated code for converting between crisp and fuzzy is fast and easily incorporated into embedded applications (see Listing 2).

Central to the manipulation of fuzzy variables are fuzzy-logic operators that parallel their Boolean-logic counterparts. These operators—f_and, f_or, and f_not—can be defined as three macros to most embedded-system C compilers as you see in Listing 3.

FUZZY EXPRESSIONS

Fuzzy expressions describe at a higher level how actions are to be performed. Implementing many of these fuzzy rules in an application results in a behavior.

This statement can read literally and evaluated with considerable precision:

```
IF Speed IS FAST AND Radar IS
 DETECTED
THEN Brake IS ON;
```

Both Speed and Radar can be measured as crisp values. Radar-detector relevance can be measured by the rate of little beeps the detector emits as you pass the industrial park or by the loud tone you get when the cop points the radar gun directly at your car.

Listing 2—*Here's an implementation example of degree of membership of FAST on a typical 8-bit microcontroller.*

```
0050                          unsigned int Speed; /* Crisp value
                                of Speed */
                              unsigned char Speed_FAST (unsigned
                              int__CRISP)
0051                          {
0100 F5 51   MOV   $51,A
0102 D3      SETB  C          if (__CRISP < 60) return(0);
0103 94 3C   SUBB  A,#3C
0105 40 02   JC    $010B
0107 E4      CLR   A
0108 22      RET
                              else {
010A E5 51   MOV   A,$51        if (__CRISP <= 80)return(((__CRISP
                                - 60) * 12) + 7);
010B D3      SETB  C
010C 94 50   SUBB  A,#50
010E 40 08   JC    $011A
0110 E5 51   MOV   A,$51
0112 D3      SETB  C
0113 94 3C   SUBB  A,#3C
0115 31 46   ACALL $146
0117 24 07   ADD   A,#07
0119 22      RET
                                else {
011A 78 FF   MOV   R0,#FF         return(255); } } }
011C 22      RET
```

crisp values to the fuzzy range between fuzzy 0 and fuzzy 1. Fuzzy expressions are also evaluated between fuzzy 0 and fuzzy 1.

The fuzzy result of the controlling expression determines the degree of braking required. A brake value of fuzzy 0 is no brake pressure, whereas a value of fuzzy 1 is full braking. It's important to note that many of the linguistic conclusions are a result of the general form of the above equation.

FROM CRISP TO FUZZY

Computations performed in the fuzzy domain must consistently translate logical operations in the crisp domain to degrees of membership in the fuzzy domain. Much of this article is devoted to practical implementations of developing a degree of membership from the crisp data.

The simplest translation is converting basic Boolean 0 and 1 data to fuzzy 0 and 1. This macro enables Boolean-logical comparisons to be made in fuzzy-logic expressions:

```
#define logical_to_fuzzy(a)
   ((a)?F_ONE:F_ZERO)
```

This C macro converts a logical value to a fuzzy 0 or fuzzy 1. It simply tests for logical 1 or 0 and replaces the Boolean-logical value with a fuzzy 1 or 0 (i.e., F_ONE, F_ZERO).

First, you replace the crisp equality comparison with a fuzzy comparison that accounts for a particular range of data. Then, you base a comparison on three data values—the comparison point, the range until the comparison has failed (delta), and the current variable value.

Delta is the distance to a value where the current comparison ceases to be important. Consider for a moment the definitions in Listings 4a–f. In each case, the delta value returns a fuzzy 0 or fuzzy 1, and any further deviation from the center point does not change the result.

Most people have a good sense of the delta required by an application. Figure 2 shows a Fuzzy Equal relationship to the crisp domain. As Listing 4a shows, the definition of Fuzzy

Listing 3—*These linguistic variables and operators written in C define typical Boolean operations.*

```
#define f_one         0xff
#define f_zero        0x00
#define f_or(a,b)     ((a) > (b) ? (a) : (b))
#define f_and(a,b)    ((a) < (b) ? (a) : (b))
#define f_not(a)      (f_one+f_zero - a)
```

Equal can be easily implemented on most small microcomputers.

This technique can be extended to normal arithmetic comparisons. In the following example, Fuzzy Not Equal has the same membership as F_NOT (F_EQ(v)). The definition of F_NOT is:

```
#define F_NOT(a) (F_ONE-(a))
```

All of the normal crisp identifiers can be used in the fuzzy domain. Membership functions for each of the normal crisp comparisons are given in Listings 4b–f.

The implementations for the fuzzy numerical comparisons all use simple linear functions to calculate the degree of membership. The merits of using some form of exponential functions on small eight-bit microcontrollers is attractive.

There hasn't been a universal consensus, but power functions can express emphasis. For instance, we might say a long fly ball is NEARLY a home run and a ball that traveled to the warning track is VERY_NEARLY a home run.

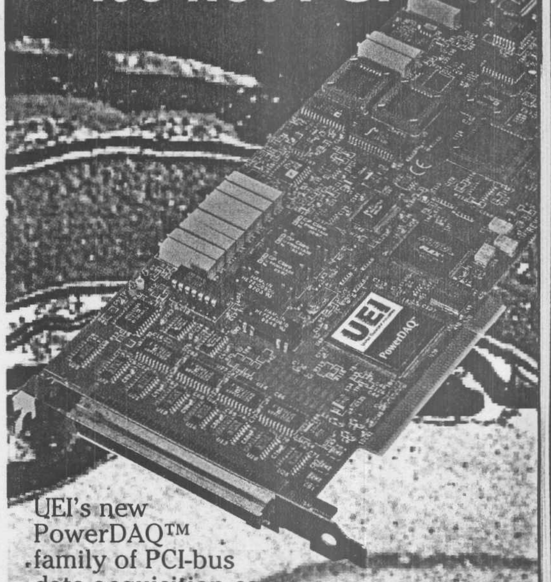
Most exponential functions can be implemented as the sum of a series. For example, a reasonable implementation for VERY_NEARLY can be made with the definitions in Listings 5a and 5b.

The linguistic function has two slopes with an intersection halfway from delta to the setpoint. Exponential functions can be implemented by degrees, where a single linear function has a degree of 0 and VERY_NEARLY (as just described) has a degree of 1. Such

Listing 4—*All normal arithmetic comparisons can be made in the fuzzy domain. Using fuctions such as these enables the developer to make close comparisons and easily mix linguistic operations with crisp ones. This effectively adds a linguistic data type to a developer's tool box.*

```
typedef DOMtype unsigned char

a) DOMtype F_EQ(v,cp,delta) {
     long m = ABS(cp - v);
     if (m > delta) return(F_ZERO);
       return((m/delta) * (F_ONE - F_ZERO)); }

b) DOMtype F_NE(v,cp,delta) {
     long m = ABS(cp - v);
     if (m > delta ) return(F_ONE);
       return(F_ONE - ((m/delta) * (F_ONE - F_ZERO))); }

c) DOMtype F_LT(v,cp,delta) {
     if (v < (cp - delta)) return(F_ONE);
     if (v < cp)
       return(F_ONE - ((cp - v/delta) * (F_ONE - F_ZERO)));
     else return(0); }

d) DOMtype F_LE(v,cp,delta) {
     if (v < cp) return(F_ONE);
     if (v < (cp + delta))
       return(((v - cp)/delta) * (F_ONE - F_ZERO));
     else return(0); }

e) DOMtype F_GT(v,cp,delta) {
     if (V < CP) return(F_ONE);
       return(F_ONE - (((v - cp)/delta)* (F_ONE - F_ZERO)));  }

f) DOMtype F_GE(v,cp,delta) {
     long m = ABS(cp - v);
     if (V > CP) return(F_ONE);
       return(F_ONE - ((m/delta) * (F_ONE - F_ZERO))); }
```

If it's not PowerDAQ™ it's not PCI



UEI's new PowerDAQ™ family of PCI-bus data acquisition cards gives you the latest and most complete implementation of the PCI specification. They include the features that other PCI data acq. manufacturers "forgot":

- Motorola 66 MHz on-board processor supervises operations
- Bus-mastering data transfer with scatter/ gather DMA
- 1 MHz A/D sample rate
- Continuous, multi-channel streaming to system memory
- Simultaneous analog and digital input and output
- Clean 32-bit drivers for Windows 95 & Windows NT
- DLL library supplied with each board (includes source code)

Call today for your free catalog:

(800) 829-4632

UEI

United Electronic Industries

10 Dexter Ave, Watertown, MA 02172

Tel: (617) 924-1155
Fax: (617) 924-1441

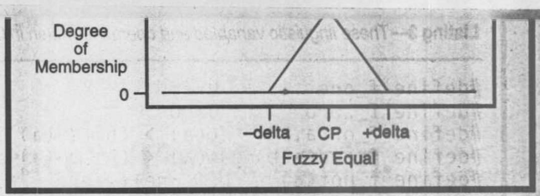
internet: www.ueidaq.com
E-mail: sales@ueidaq.com

#118

with reasonable compilers.

Degrees of membership normalize the structure of the data to be scaled between fuzzy 0 and 1. Exponential functions can be applied in series with degrees of membership to emphasize or deemphasize membership functions in application code.

matches. The better the match, the higher the degree of membership.



HEDGE MODIFIERS

The C macro `HEDGEmodifier` specifies the hedge `very` as being of order 1 (single break in power function):

```
#define very(a)
   HEDGEmodifier(1,a)
```

Listing 6a defines a hedge of order 1. Note that this function is easily implemented on most small microcomputers.

Negative hedge modifiers describe such concepts as `ROUGHLY`. I might say a certain man is about 45 and be referring to someone within a year or two of 45. I can then describe a broader range of individuals as `ROUGHLY.ABOUT` 45.

```
#define ROUGHLY(a)
   HEDGEmodifier(-1,a)
```

A hedge modifier of order –1 is defined as in Listing 6b. As an alternative, hedge functions can be implemented as simple table look-up functions with at most some linear interpretation.

Hedge functions may be used with any membership function and may be cascaded to strengthen emphasis. Two hedges of order 1 in series yield a hedge of order 3. I haven't needed to implement hedges other than 1, 0, and –1.

Hedge functions permit the application designer to formally add emphasis. Here, hedge functions have a common mathematical basis and an application-specific meaning. And, they smooth out the roughness in descriptions.

The quest for improved accuracy in the last twenty years has mostly been a quest for improved numerical relevance. The fuzzy domain always focuses on the relevant data and limits to fuzzy 1 or 0 when the data ceases to be relevant.

IN THE REAL WORLD

The techniques I've described go a long way toward describing a problem in an effective and natural way. Crisp comparisons are necessary in some contexts. But in the last few years, we've discovered many more applications for fuzzy logic than we had previously imagined.

I am primarily in the code-creation business and recognize that most application software is written on conventional computers with conventional instruction sets. All of the definitions in this article can be implemented on even basic eight-bit micros using acceptably few instructions, typical shifts, adds, and comparisons—things that eight-bit micros do well.

It's practical to build fuzzy-logic control systems in consumer appliances. Against conventional implementation techniques, fuzzy logic competes well. In general, my experience has been that fuzzy logic is less math intensive and more logic intensive than traditional implementations.

Listing 5—*Exponential functions can be implemented by degrees. `NEARLY` and `VERY_NEARLY` represent different levels of hedge functions.*

```
a) DOMtype NEARLY (long v,sp,delta) {
     long m = sp - v;
     if (m < 0) return (F_ZERO);
       return((d/m) * (F_ONE - _ZERO));}

b) DOMtype VERY_NEARLY(long v,sp,delta) {
     return((nearly(v,sp,delta) +  nearly(v,sp,(delta/2)) / 2);}
```

Listing 6—*Here's the general form for hedge functions for order 1 and order –1. Hedge functions can be cascaded to increase the extent of a condition.*

```
a) hedge order 1 translation is
     if (in < 0.5)
       return (in >> 1)
     else
       return (0.25 + (in - 0.5) * 1.5);

b) hedgemodifier  order -1 translation is
     if (in < 0.5)
       return (in * 1.5
     else
       return (0.5 + (in - 0.5) * 0.5);
```

Walter Banks is president of Byte Craft Limited, a company specializing in software tools for embedded microprocessors. His interests include highly reliable system design, code-generation technology, programming language development, and formal code-verification tools. You may reach him at walter@bytecraft.com.

SOURCE

Fuzz-C, C 8051 compiler
Byte Craft Limited
421 King St. N
Waterloo, ON
Canada N2J 4E4
(519) 888-6911
Fax: (519) 746-6751
www.bytecraft.com

I R S

407 Very Useful
408 Moderately Useful
409 Not Useful